TASK ALLOCATION IN A DISTRIBUTED COMPUTING SYSTEM

Walter D. Seward
Air Force Institute of Technology
Department of Electrical and Computer Engineering
Wright-Patterson AFB, OH 45433

ABSTRACT

Distributed computing offers the potential for improved system's performance for many applications. Critical to the realization of this performance improvement is a methodology for task allocation which considers both the application requirements and the system architecture. This paper examines a conceptual framework for task allocation in distributed systems and discusses application and computing system parameters critical to task allocation decision processes. The paper addresses task allocation techniques which focus on achieving a balance in the load distribution among the system's processors. That is, equalization of computing load among the processing elements. Examples of system performance are presented for specific applications. Both static and dynamic allocation of tasks are considered and system performance evaluated using different task allocation methodologies.

INTRODUCTION

Recent advances in the development of microcomputer systems has increased interest in the use of distributed computing systems. Certain applications such as the Battle Management and Command, Control, and Communications (BM/C3) requirements of a strategic defense system appear naturally as distributed computational systems.[1] In addition, despite the impressive speed of the current generation of computers, their architecture limits them to a mostly serial approach to computation, and limits their usefulness for problems that are computational intensive and which may require processing speeds upwards of 100 million operations per second. Physical limits suggest that these traditional, serial architectures offer little hope of large performance improvements. Distributed and parallel processing systems offer an opportunity for improved system performance, reliability and flexibility. Critical to the realization of increased system capabilities is an effective means of allocating the processing tasks among the system's computing resources. Without an effective scheme of task allocation, the performance of the distributed system can be degraded to something less than that of one of the system's single processors.

This paper presents preliminary results of research performed to analyze the performance of task allocation methodologies for a distributed computing system. For this paper, distributed processing is considered as a special case of parallel processing where the processing elements are loosely coupled and any exchange of information and control of the system must take place via an interconnection structure instead of by means of shared memory. However, the fundamental approaches used in this work do not preclude use with a tightly coupled parallel processing system. The appli-cation program used in this analysis was that of Integer Linear Programming (ILP). ILP is representative of a class of algorithms which is applicable to the solution of several problems within the BM/C3 environment of strategic defense. Results of this analysis show that dynamic task allocation can provide significant performance improvement. Also discussed are planned extensions to this research.

PERFORMANCE OF DISTRIBUTED SYSTEMS

Evaluation of alternative system implementation must be based a relevant metrics. Because increased speed of computation is one of the primary reasons for using parallel systems, the speed of the parallel algorithm is one of the most important parameters for system evaluation. The most frequently used measures of parallel and distributed system performance are

speed-up and efficiency. These measures are defined as follows: [2]

$$S = \frac{\text{worst-case running time of fastest known sequential algorithm for the problem}}{\text{worst-case running time of parallel algorithm}} \quad (1)$$

$$E = \frac{\text{worst-case running time of fastest known sequential algorithm for the problem}}{\text{cost of parallel algorithm}} \quad (2)$$

where,

S = Speed-up
E = Efficiency

and the cost of parallel algorithm is defined to equal the product of the parallel running time and the number of processors used.

The ratio of single processor time to parallel system time can be expressed as $S = T_1/T_N$ where T_1 is the

single processor time and the computation time using N processors, T_N, is determined by the following formula:

$$T_N = T_S + \text{Max}\ (T_C) + T_W \quad (3)$$

where

T_N = Computation Time for N processors

T_S = Start-up Time

Max (T_C) = Time required for last busy processor to complete its computation

T_W = Wind-Down Time.

Start-up time measures the time that is required to initialize the system before any computations can begin. Wind-Down time refers to the time required to collect from the various system processors and analyzing or tabulating into a final product. The time required for the last busy processor to complete its computations includes several elements. Included is all of the time during which this processor and all other processors in the system were actually performing useful computations. Also included is any idle time, time required to communicate with other processors, and time required to perform overhead functions built into the parallel algorithm.

Maximum speed-up, which is the overriding objective for most parallel processing applications, occurs when the parallel processing time is minimized. The elements of the equation for the parallel processor time are not independent. The algorithm, the architecture and their implementations determine the specifics of the relationships among these parameters. However for applications of interest the predominant factor, by orders of magnitude, is the computation time. Thus, speed-up is maximized when $\text{Max}(T_C)$ is minimized.

Balanced Computational Load

There are three rules of thumb for minimization of $\text{Max}(T_C)$ and, therefore, maximizing the performance of parallel processing systems: [3]

(1) Distribute the computation load evenly;

(2) Maximize the computation time to communication time ratio;

(3) Minimize communication distance.

These rules of thumb, unfortunately, may conflict. For example, in order to maintain a balanced computational load additional communications overhead may be required. Thus $\text{Max}(T_C)$, which includes both computation and required interprocessor communications, must be considered in its entirety. Just maintaining a balanced workload with out consideration of associated overhead and memory system costs may result in decreased system performance. This is the task allocation issue being addressed in this research.

TASK ALLOCATION

Concepts and techniques for task allocation or task scheduling have evolved from the considerable body of work on job-shop or assembly-line problems. [4,5] Work in this area has been has been extensively documented in management science, operations research, and computer science and engineering literature. The two fundamental approaches to task allocation are static and dynamic. Each of these techniques has advantages and disadvantages which are a function of the application and the system architecture.[6]

Static Allocation

Static allocation involves the a-priori determination of the allocation of tasks. This method is the least complex and requires the least overhead

of the system during its operation. In systems where complete knowledge of the computational requirements of the application tasks and the system's operational characteristics are known, static allocation of task can be the most efficient. Minimal resources are required during run-time while maintaining high levels of efficiency. But, solution of the static allocation problem in these cases generally requires solution by means of an off-line integer programming problem or other computational intensive combinatorial algorithm. Furthermore, when knowledge of the system operational characteristic while dedicated to a specific application, or if the exact computational requirements are not known, static allocation is not generally effective. In these cases, all advantages of parallelism in the problem structure and architecture may be lost and the total computational load assigned to a small subset of the available processors.

Dynamic Allocation

Dynamic allocation of tasks is an attempt at maintaining the most efficient use of the system's computational resources by adapting, at run-time, the load distribution to the demands of the computational tasks. It involves assessing the availability of each processing element to accept additional work and reallocation of the total system computational load so that a stated performance metric is optimized. The process of obtaining an optimal result is itself a large-scale combinatorial optimization problem, and therefore dynamic allocation techniques are most often based on heuristic procedures. Moreover, the difficulty of the task is increased by lack of precise information about the application program's requirements and the computational characteristics of the system. These factors must, in general, be estimated using some method of heuristic or statistical technique. The objective of any technique of dynamic allocation is to optimize the use of the system communications, computation, and memory resources, but this requires minimization of overhead for execution of the allocation routine, which requires specific knowledge of the distributed system architecture.

DISTRIBUTED SYSTEM ARCHITECTURE

Evaluation of the system's performance require consideration of both communications and computational dependent parameters specific to the system implementation. The performance of any dynamic task allocation method depends upon not only the application, but these characteristics of the computing system's architecture.

The distributed architecture system used for this study was the Intel iPSC system.[3] This system is based on a hypercube topology where each node of the topology is a processor with its own memory and communications capability. The system operation is based on a process model of computation supported by message passing. A hypercube or binary n-cube is a network of 2^n processors each with direct communications with its n neighbors. The number n defines the dimension of the cube. Figure 1 illustrates a cube of dimension 4.

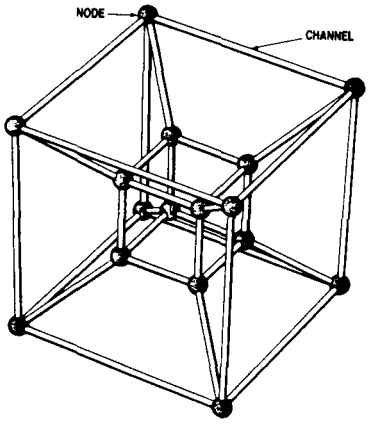

Figure 1 Dimension Four Hypercube.[3]

The iPSC can be configured with modules of 16, 32, 64 or 128 interconnected nodes. The processors which form the elements of the hypercube are called nodes. Each node is numbered by an n-bit binary number between 0 and 2^n-1. Node 0 is connected to a separate processor called the "cube manager" which provides the input and output link between the nodal processors and the outside world.

Communications within the cube is based on a static routing algorithm which routes messages by a minimum number of hops along the edges of the cube. Although there is support for communications processing on each node, the nodal processor, an Intel 80286 chip, must perform some of the tasks associated with forwarding a message. Hence the pattern of communications within the application's implementation may have a significant impact on the overall performance.

The iPSC system provides support for software developed in 80286 assembly language, C, and Fortran 77. The algorithms developed for this paper were implemented in Fortran 77. This language was chosen because compiled Fortran 77 is faster than C on the iPSC.

INTEGER LINEAR PROGRAMMING

The specific problem analyzed in this research is a distributed computer algorithm for the solution of Integer Linear Programming problems. In matrix representation, Integer Linear Programming (ILP), is an optimization problem which consists of finding a vector $\underline{x}$ which maximizes (minimizes) a linear objective function $\underline{C}*\underline{x}$ subject to a set of linear constraints $\underline{A}*\underline{x} = \underline{b}$, $\underline{x} \geq 0$, and $\underline{x}$ integer. $\underline{A}$ is an m by n matrix, $\underline{C}$ is a 1 by n row vector, and $\underline{b}$ and $\underline{x}$ are n by 1 column vectors all of which are integer.[7] Such problems are frequently posed as problems with inequality constraints which must be transformed to the required format by adding "slack" and "surplus" variables.[7] Although the problem is easily expressed it is extremely difficult to solve in practice. It has been shown, in fact, that ILP is NP-complete.[8]

Among the techniques frequently used to solve the ILP is an enumerative technique called the branch-and-bound.[5] A naive approach to enumerative solution of an ILP involves the explicit examination of each possible integer vector to determine the one which results in maximization (minimization) of the objective function. However, even for relatively small ILP's the number of possibilities, while finite, becomes excessively large. Consider for example an ILP with ten unknown variables each of which has an integer range of zero to ten. There are more than ten billion possible solutions to this problem. Explicit enumeration of the possible solutions quickly becomes overwhelming.

The branch and bound method of solution of the ILP uses a combination of explicit and implicit enumeration to reduce the total number of permutations which must be examined. The branch and bound technique can best be described in terms of a search tree. The height of the tree is determined by the number of integer variables, n, in the ILP. A path from the root node to a leaf node which satisfies all problem constraints corresponds to a feasible integer solution. At a given level of the tree the nodes correspond to specific integer values for one of the problem variables. Figure 2 illustrates a search tree for a three variable ILP where the x_1 has a domain of {0,1,2} and x_2 and x_3 each have a domain of {0,1}.

Initially the ILP algorithm performs an unconstrained solution to the linear programming problem which results if no integer constraints are applied. If no feasible solution exists to the unconstrained problem, then no solution exists for the ILP. The value of the objective function for the feasible solution to the unconstrained problem represents an upper (lower) bound on the ILP objective function for the maximization (minimization) problem.

The branch and bound algorithm used in this research performs a depth first search of the enumeration tree successively constraining the integer variables to specific values. As each variable is constrained, the value of the objective function for the feasible partial solution is compared with the existing lower (upper) bound to determine if further search along that path is justified. The lower (upper) bound is determined by the maximum (minimum) objective function value for feasible integer solution to the ILP. As a new larger (smaller) bound is determined it replaces the existing lower (upper) bound. The bound defined by the feasible integer solutions determines the criteria for eliminating nodes from the search tree before extending them. Any partial solution which falls below (above) the feasible solution bound cannot represent an improved solution to the maximization (minimization) problem.

Distributed Computation of ILP

The ILP was implemented on the Intel iPSC distributed computing system in the following fashion:

1. One node of the hypercube is designated the responsibility of coordinating the computation of the other processors in the system. This controller node partitions the problem space, assigns blocks of the partition to independent processing elements, collects intermediate results and shares these results with the other processors, and maintains the status of the progress of the solution.

2. The "worker" nodes of the system perform the ILP algorithm on assigned blocks of the problem space partition. These nodes inform the controller node of improved results, current status in solving the assigned partition, and if they are idle.

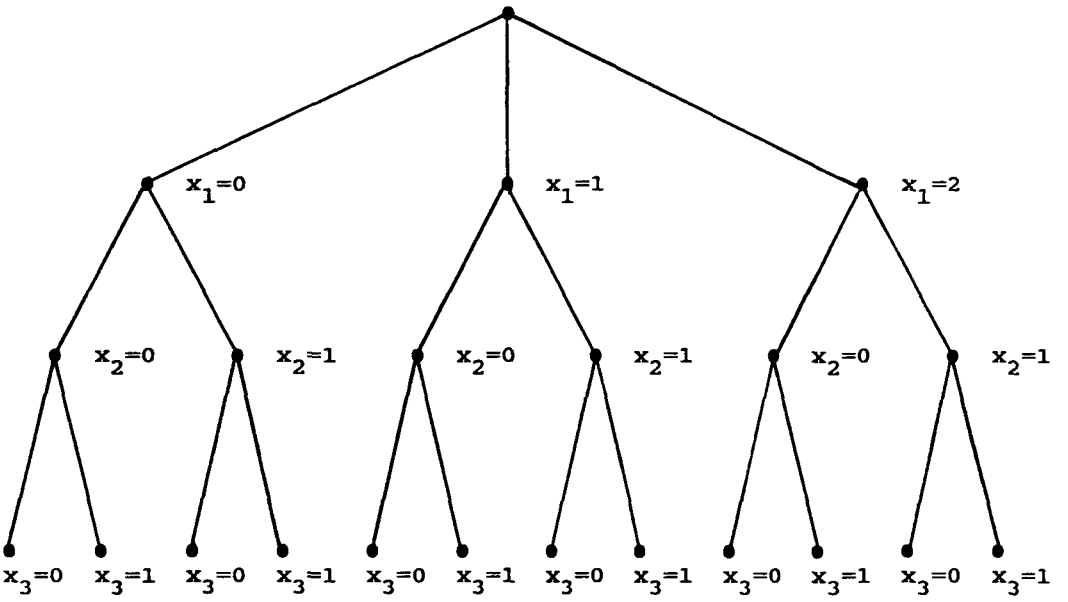

Figure 2 Example Enumeration (Search) Tree

Computation performed by the worker nodes includes the branch-and-bound algorithm. These nodes send to the controller node the objective function value for each improved integer feasible solution. This information is shared with all other nodes so that more rapid implicit enumeration is possible. These nodes also provide the the controller node with a status vector which represents their progress in solving assigned subproblems of the ILP problem.

RESULTS AND ANALYSIS

This section presents the results of experiments performed using the Intel iPSC to compute the ILP for a group of problems which range in complexity from easy to moderately difficult. Table I contains the descriptions of these problems. All of these problems are stated in the form of $\underline{A}*\underline{x} \leq \underline{b}$. The performance of these ILP problems using differing configurations of the iPSC and differing methods of task allocation are discussed in the following sections.

Baseline Performance

A baseline of system performance for the four problems is presented in Table II. The processor time required to perform a sequential version of the ILP computation using a VAX 11/785 and a single node of the iPSC are compared. The ratio of the VAX time to the node processing time is seen to remain relatively constant at approximately 0.285. As can be seen for Problem 4, the time required on either system is excessive for most environments. These results are based on the depth first search of the enumeration tree and are dependent upon the speed with which the lower bound converges to the optimal solution. The longer it takes to converge the larger the problem space which must be explicitly evaluated.

Table III presents the "best case" times for these problems using a single node of the iPSC. These results were obtained by setting the lower bound to the previously determined optimal value and allowing the search algorithm to perform maximal implicit enumeration. The results show a substantial speed-up for all but Problem 2. Problem 2 does not show the same improvement because its optimal solution is encountered early in a depth-first search. Hence implicit enumeration quickly dominates the enumeration scheme even for the sequential implementation.

The results obtained by this method are not representative of normal performance. They depend upon a-priori knowledge of the answer. These results are included so that the parallel system performance can be compared with the best possible sequential performance for the specific algorithm used.

Static Partition and Allocation

Except for those applications where knowledge of the problem space is complete enough for static partition and allocation to provide sufficient

Table I

Integer Linear Programming Problems Used for System Evaluation

Problem 1

Objective Function = $4x_1 + 8x_2 + 5x_3$

Variables

x1	x2	x3	b
1	2	3	18
1	4	1	6
2	6	4	15

Problem 2 [7:370]

Objective Function = $x_3 + x_4 + x_5$

Variables

x1	x2	x3	x4	x5	b
2	3	1	2	2	18
3	2	2	1	2	15
-6	0	1	0	0	0
0	-7	0	1	0	0

Problem 3 [7:371]

Objective Function = $x_7 + x_8 + x_9 + x_{10} + x_{11} + x_{12}$

Variables

x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	b
9	7	16	8	24	5	3	7	8	4	6	5	110
12	6	6	2	20	8	4	6	3	1	5	8	95
15	5	12	4	4	5	5	5	6	2	1	5	80
18	4	4	18	28	1	6	4	2	9	7	1	100
-12	0	0	0	0	0	1	0	0	0	0	0	0
0	-15	0	0	0	0	0	1	0	0	0	0	0
0	0	-12	0	0	0	0	0	1	0	0	0	0
0	0	0	-10	0	0	0	0	0	1	0	0	0
0	0	0	0	-11	0	0	0	0	0	1	0	0
0	0	0	0	0	-11	0	0	0	0	0	1	0

Problem 4 [9:236]

Objective Function = $-10x_1 + 7x_2 - x_3 + 12x_4 - 2x_5 - 8x_6 + 3x_7 + x_8 - 5x_9 - 3x_{10}$

Variables

x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	b
-3	-12	8	1	0	0	0	0	8	-2	8
0	1	10	0	5	-1	7	1	0	0	13
-5	-3	1	0	0	0	0	-2	0	-1	-6
5	3	-1	0	0	0	0	2	0	1	6
0	0	4	-2	0	5	1	-9	2	0	8
0	9	0	-12	7	-6	0	2	15	3	12
8	5	-2	-7	1	0	-5	0	10	0	16

performance, static techniques tend to be unsuccessful. System performance using static partition and allocation of the ILP problem space depends on the application program size and the created subproblem (blocks of the partition) granularity.

The static algorithm partitions the problem space into one or more subproblems per processor. The subproblems are allocated first-in-first-out (FIFO) to any idle processor until the subproblems created in the original partition are depleted. Once the list is empty, processors remain idle after completing their assigned subproblems. Insufficient numbers of subproblems quickly results in system speed which approaches that of a single processor. The larger the problem space the more likely static techniques will not substantially improve performance. This occurs because prevention of single processor degeneration requires very large numbers of subproblems which in turn results in high communications to computation ratios.

For example, even when the solution of Problem 4 is tuned by providing the optimal solution the overall solution time (with 31 nodes and 2000 subproblems per node) is only 15 percent better than the single processor performance, and the computational load is distributed such that one processor is always busy while the other 30 processors are busy only 4.6 percent of the time.

Table IV shows the results of static partition and allocation for varying problems with the numbers of subproblems fixed at 250 per node. Table V illustrates the performance of static

methods using a range of partition sizes on one of the problem sets. Clearly general techniques must involve methods other than static allocation.

Dynamic Allocation Procedures

The dynamic allocation procedures implemented in the algorithms used in this research incorporate several heuristic techniques. Maximum parallelism of the computation results if all processors initial idle time is minimized. Toward this end, the initial static partition and allocation is performed as specified for the static algorithm. Once the initial set of subproblems is depleted, any processor that becomes idle notifies the control node of its status. The control node is responsible for finding additional work, if it exists, and interrupting a busy processor so that the remaining work can be partitioned and reallocated. It is the procedures used to determine which processor to interrupt and how to partition that processor's work load that incorporate heuristics.

The decision as to which processor to interrupt is based on which processor has been busy for the longest time. It is assumed that this processor has not been able to perform implicit enumerations on most of its subproblem, and therefore, contains many feasible integer solutions. Once the processor is interrupted, the number of subproblems to form must be decided. To form only one subproblem is counter productive. The time lost to overhead of interrupt and partition will be repeated each time a processor becomes idle. Instead, the partition creates many subproblems which are used to create a FIFO problem queue, as is done for the initial partition, therefore no other processors are interrupted until the queue is empty. The number of subproblems to create for each interrupt is bounded below by a user specified minimum and calculated using a multiplicative factor determined by the partial solution object function value and the current maximum feasible object function. The closer to the maximum the more subproblems created. Finally, as the solution is approached the possibility exists for thrashing to occur. To prevent repeated interrupts of processors, a minimum busy time can be specified by the user.

Performance of Dynamic Task Allocation

The results of experiments using the dynamic task allocation method described above are presented in Tables VI and VII and Figures 3 and 4.

The results shown in Table VI illustrate the performance of the dynamic partition and task allocation algorithm on the four test problems. The results cited are the best measured performance for each problem as determined by the algorithm parameters of number of initial blocks in the partition, minimum number of blocks in each partition of an interrupted subproblem, and the minimum busy time. There is a clear performance improvement over the static allocation results shown in Table IV. Moreover, the performance improvement increases as the problem complexity increases.

As shown in Table VI, the speed-up for Problem Four exceeds linear, with respect to the number of processors, when compared to the single processor time without a-priori knowledge of the solution. With a-priori knowledge of the solution, the speed-up is less than linear but substantial. Further analysis of the the super-linear performance is in order.

Based on the serial processing algorithm implemented in this study, super-linear speed-up is achievable. However, could the performance of this algorithm be improved so that it approaches the performance measured with a-priori knowledge as shown in the "best case" performance? If such an optimal algorithm can be determined, then it should be applied. Improvements may come from different search techniques or improved implementation techniques; however, it is questionable if such an algorithm can be developed for the general case.

The advantage of the parallel solution of the ILP using a branch and bound algorithm is that the knowledge essential to rapid solution of the problem, the bounds, are determined more quickly. These results are broadcast to all computing elements, thereby accelerating their performance. If a sequential algorithm can be optimized, then each of the nodes could also use the algorithm. The performance improvement using multiple processors would approach linear, and differ only by the overhead associated with communications, start-up, and wind-down.

Table VII illustrates the performance of the dynamic scheduling algorithm applied to Problem Four for varying numbers of nodes. The speed of performance improves approximately linearly with the number of processors. These results were based on the same number of blocks in the initial partitions, the same number of blocks in the subsequent partition of

subproblems, and the same minimum busy time.

Figures 3 and 4 illustrate the criticality of the dynamic task allocation parameters. There exists a clear indication that specific values provide significant performance improvements. Moreover, certain values for the parameters result in a significant degradation in the performance. How the "optimal" parameters are determined is the subject of on-going research.

SUMMARY

The results illustrate that reasonable performance from a distributed computing system requires some method of dynamic task allocation. Such an allocation scheme must consider not only the application but the characteristic of the system architecture used for the implementation. Specifically the processing speed, the communication subsystem characteristics, and the memory requirements.

The results of pervious experiments demonstrates the need for an adaptive algorithm to "tune" the application algorithm's performance to the system architecture. On-going research is examining the development of automated techniques for generating the critical task allocation parameters. Under consideration are methods of stochastic estimation and heuristics for determining the best combination of allocation algorithm parameters. In addition, further work is being performed to improve the serial algorithm by consideration of different search strategies.

TABLE II

Serial Processing Time
(in Seconds)

PROBLEM NUMBER	INTEL iPSC (1 NODE)	VAX 11/785
1	11.1	3.6
2	49.8	14.8
3	4067.3	1193.6
4	108335.9	32052.0

TABLE III

"Best Case" Serial Processing Time
(in Seconds)

Problem Number	INTEL iPSC (1 Node)
1	0.795
2	49.685
3	750.295
4	3222.355

Table IV

Static Partition and Allocation
(31 Nodes)

Problem Number	Solution Time (Seconds)	Speed Up
1	4.12	2.69
2	45.81	1.09
3	1689.31	2.41
4	108289.00	1.00

Table V

Problem 3
Static Partition and Allocation
(31 Nodes)

Number Blocks Per Node	Solution Time (Seconds)	Speed Up
100	1691.6	2.4
200	1689.5	2.4
500	1664.3	2.5
600	1736.0	2.3
800	1736.1	2.3
1000	1691.7	2.4
1500	1691.6	2.4
2000	1689.5	2.4

Table VI

Dynamic Partition and Allocation
(31 Nodes)

Prob. #	Solution Time (Sec.)	T_1/T_N	"Best T_1"/T_N
1	3.22	3.45	0.24
2	13.86	3.59	3.58
3	488.23	8.33	1.57
4	155.09	698.54	20.78

Table VII

Problem 4
Dynamic Partition and Allocation

Nodes	Solution Time (Sec.)	T_1/T_N	"Best T_1"/T_N
4	1233.9	87.8	2.61
8	664.7	163.0	4.85
16	373.7	289.9	8.62
31	155.0	698.94	20.78

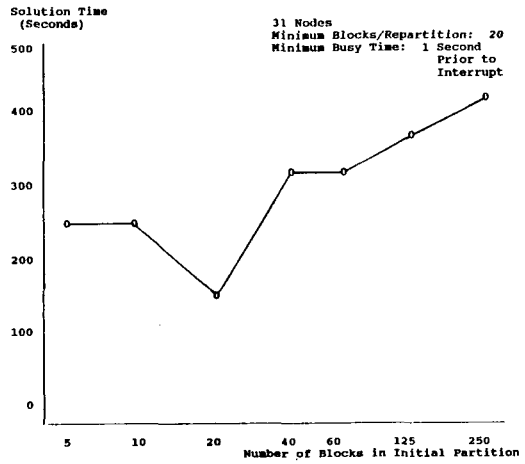

Figure 3 - Problem 4: Solution Time vs Initial Blocks

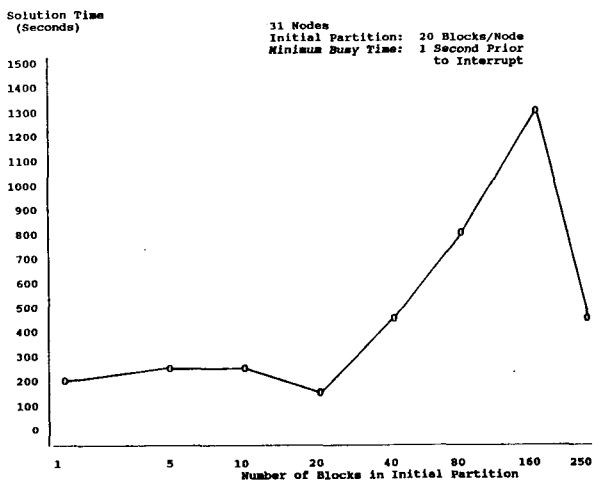

Figure 4 - Problem 4: Solution Time vs Initial Blocks

ACKNOWLEDGEMENTS

This research is supported in part by the Strategic Defense Initiative Organization. Also, I gratefully acknowledge the extensive software development and technical development of Captain Paul Bailor a PhD student at the Air Force Institute of Technology.

REFERENCES

1. Seward, Walter D. and Nathaniel J. Davis IV, "Opportunities and Issues for Parallel Processing in SDI Battle Management/C3," Unpublished report Presented at the AIAA Computers in Aerospace V Conf., Oct 1985.

2. Akl, Selim, PARALLEL SORTING ALGORITHMS, Academic Press, Orlando, FL, 1985, pg 8-9.

3. ..."Intel iPSC Concurrent Programming Workshop Notes," Intel Scientific Computers, Beaverton, OR, 16-20 June 1986.

4. Coffman, E. G. and Denning, P. J., OPERATING SYSTEMS THEORY, Prentice-Hall, Englewood Cliffs, NJ, 1973.

5. Kohler, W. H. and Steiglitz, K., "Enumerative and Iterative Computational Approaches," COMPUTER AND JOB/SHOP SCHEDULING THEORY, E. G. Coffman ed., John Wiley and Sons, NY, 1976, pg 229-287.

6. French, S., SEQUENCING AND SCHEDULING, Halsted Press, NY, 1982.

7. Garfinkel, R. S. and Nemhauser, G. L., INTEGER PROGRAMMING, John Wiley and Sons, NY, 1972.

8. Papadimtriou, C. H. and Steiglitz, K., COMBINATORIAL OPTIMIZATION: ALGORITHMS AND COMPLEXITY, Prentice-Hall, Englewood Cliffs, NJ, 1982.

9. Salkin, Harvey, INTEGER PROGRAMMING, Addison-Wesley, Reading, MA, 1975.